MacsBug Documentation

MacsBug is the Macintosh 68000 Debugger. This chapter describes the theory and operation of MacsBug. It also describes the syntax of commands accepted by MacsBug.

Contents:

Note: This chapter is specific to MacsBug 5.1 for Macintosh. Some information does not apply to earlier versions of MacsBug, which had different features or ran on different machines, such as the Lisa.

About MacsBug

MacsBug is a line-oriented single-Macintosh debugger. It resides in RAM along with the application being debugged. The capabilities of MacsBug include:

- the ability to display and set memory and registers
- the ability to disassemble memory
- stepping and tracing through both RAM and ROM
- monitoring of system traps
- display and checking of the system and application heaps

MacsBug obtains control when certain 68000 exceptions occur. You can then examine memory, trace through the application, or set up break conditions and execute the application until those conditions occur.

MacsBug 5.1 works with the following hardware configurations:

- 512K to 4 MB of RAM
- 64K Macintosh or 128K Macintosh Plus ROMs
- Motorola 68000 or 68020 processors

- Motorola 68881 floating-point coprocessor

MacsBug 5.1 does not work on the Macintosh XL; if it accidentally finds its way onto a Macintosh XL, it simply will not install itself. Macintosh XL users should use MacXLBug.

Installing MacsBug

MacsBug is not a normal Macintosh application. Instead, MacsBug installs itself once at boot time and remains active until shutdown. This occurs if the following conditions are met:

1) MacsBug exists and is named exactly "MacsBug". (An alternate name can be used by editing the boot blocks debugger name field with a standard utility like FEdit.)
2) It is on a startup (bootable) disk.
3) Is is in the current System Folder. (This is only a requirement on HFS volumes. The system folder is, by definition, the folder which contains a system file named "System" and the system file named "Finder".)

At the completion of a successful installation, the message “MacsBug installed” is displayed right below the “Welcome to Macintosh” message. The startup application is then launched as usual.

The only way to remove an active copy of MacsBug is to reboot. To override the installation of MacsBug on a one time basis, hold the mouse button down during boot and MacsBug will not be installed. To permanently override the installation of MacsBug simply rename it to a different name or remove it completely from the disk.

*Using HFS with the 64K ROM***:** If you have MacsBug on a Hard Disk 20 (HFS) boot disk on a machine with the original 64K ROM, there is a seeming conflict with the above mouse-down commands, because holding the mouse button down at boot also forces Macintosh to launch off of the floppy disk rather than switch-launching to the Hard Disk 20. However, a skillful mouser can accomplish either command simply by knowing that MacsBug is installed first, right after the “Welcome to Macintosh” hello message, and that the HFS code looks for a mouse-down only after the “MacsBug installed” message.

Theory of Operation — A Technical Aside

*Note***:** Most people can skip this section. It is not essential in order to use MacsBug but is provided as important information to those who are interested in implementing a debugger to be used with Macintosh. It also provides a better background as to how MacsBug really works.

The state of the world when MacsBug is about to be loaded is fairly complete. The interrupt system, memory manager, and ROM based I/O drivers have already been initialized by the ROM boot code. The boot code initializes the event manager, the font manager, the resource manager, and the file system. (Although the Toolbox is initialized at this point, MacsBug does not use the Toolbox.) The 'DSAT' table is loaded in and the string "Welcome To Macintosh" contained therein is displayed.

Next the loading process of MacsBug takes place as follows: First the boot blocks code reserves some space (1024 bytes) for MacsBug's own global variables. Then this code proceeds to look for the file specified in the boot blocks as described above. If the file is not found (either due to an improper location in an HFS volume or simply because the file is not on the boot volume) then the global space is deallocated and the boot process continues normally without installing a debugger.

If MacsBug is found, the data fork (not resource fork!) of the file is then loaded onto the current stack which is located right under the main screen buffer in memory.

Historical Note: For reasons related to the original Lisa Workshop, the first block (512 bytes) of the MacsBug data fork is stripped off during this loading process. In MacsBug 5.0 and later this first block is used simply for version info about the file, as it does not take up any RAM.

Then the boot code JSRs to MacsBug itself. MacsBug begins its installation process by checking to see if the mouse button is down. If it is, MacsBug aborts the installation and lets the boot process continue without installing itself. If the button is not down, MacsBug determines which kind of machine and microprocessor it is running on and configures itself accordingly.

When installed, MacsBug puts pointers to itself in many of the hardware exception vectors in addresses $0000 0000 through $0000 00FF. It then remains dormant until one of "its" exceptions occurs. The following list of exceptions to which MacsBug responds are each numbered one greater than the corresponding Macintosh System Error numbers.

Exception #	**Assignment**
2	Bus Error (rarely seen on Macintosh)
3	Address Error (not aligned to a word boundary)
4	Illegal Instruction (bit pattern not recognized)
5	Zero Divide
6	CHK Instruction (array index out of bounds)
7	TRAPV Instruction (overflow)
9	Trace (used to single step in MacsBug)
10	Line 1010 Emulator (The A-trap handler for all Toolbox traps)
11	Line 1111 Emulator (68xxx co-processor trap interface)
28	Level 4 Interrupts
29	Level 5 Interrupts
30	Level 6 Interrupts
31	Level 7 Interrupts
47	Trap $F Instruction (Used for setting programmer breaks)

68000 exception processing is described in the Motorola 68000 *Programmer's Reference Manual*.

Any time an A-Trap or other exception listed above occurs, MacsBug intercepts the trap and can thus stop or display the current state of the machine. Single stepping through 68000 instructions is possible because the Trace bit in the status register of the microprocessor can be set by MacsBug.

During installation MacsBug obtains further memory from below the main screen buffer for use as its own screen memory. (MacsBug obtains memory from the same general area in RAM as RAM disks and caching utilities. It is based on the location of BufPtr, a Macintosh global variable.) MacsBug 5.1 offers a full screen display with 40 lines saved. This display uses about 20K of memory.

The total RAM memory requirement of MacsBug 5.1 is approximately as follows :

Global space	1 KB
Screen space	20 KB
Code space	22 KB
TOTAL:	43 KB

MacsBug thus may not work with some memory-intensive applications on a 512K Macintosh. Two solutions are possible:

1) Remove MacsBug to free up about 43K of RAM
2) Add additional RAM via the Macintosh Plus Logic Board Upgrade or compatible third party hardware upgrade to 1 MB of RAM or more.

At the successful completion of the installation process an alert is posted below the "Welcome To Macintosh" screen saying "MacsBug installed". The boot process then continues by loading INIT resources from the System file.

Technical Note: The boot code looks for INITs 0-31 and JSRs to them. These 'INIT's are used to set up the keyboard maps (INITs 0 and 1), install patches (of type PTCH) to ROM code, and so on. 'INIT' 31 extends the system further by looking for any files of type INIT in the System Folder. This facility allows developers to install their own startup code without changing the System file. This could be another possible way to install yourself for debugging purposes. An 'INIT' resource in a file of type 'INIT' could be created with ResEdit to immediately drop into MacsBug, for example. The 'INIT' resource could simply be the hex string $A9FF 4E75.

Then the startup application is launched. This is typically the Finder, but the startup application can be set to any other application through the "Set Startup" menu item in the Finder.

Using MacsBug

The simplest way to get into MacsBug is to generate an exception by pressing the interrupt button. (The interrupt button is the rear button of the programmer’s switch on Macintosh or the minus key on the numeric keypad on Lisa).

To see the application screen while the debugger is active, press the tilde/opening quote key (~/ `) in the upper-left corner of the keyboard. Repeated presses toggle between the two screens, thus allowing easy viewing of both the actual code (MacsBug screen) and the results (main screen). To restore the debugger’s display, press any character key.

The best way to programmatically enter the debugger is to use the system trap called Debugger as a breakpoint in your program at the point where you want MacsBug to get control. There are two ways to use this trap. Calling trap $A9FF drops into MacsBug and displays the message "USERBRK." It then does a normal exception entry into MacsBug (unless you have toggled the DX command— see "Break Commands" below). If you want to display custom debugging information, declare and call the trap with bit 10 set ($ABFF) . When this latter trap is encountered, MacsBug assumes that the top of the user's stack has a pointer to a Pascal string. It prints out the string, displays the message "USERBRK," and does a normal exception entry into MacsBug.

Here is a summary of how to declare and use this trap from MDS Assembly Language:

Declaration

```
.TRAP       _Debugger  $A9FF   ; predefined in the file Traps.txt
.TRAP       _DebugStr  $ABFF   ; not predefined - define it  yourself
```

Example Calls

a)
```
_Debugger                ; enters MacsBug and displays USERBRK
```

b)
```
STRING PASCAL            ; asm directive to make sure to push a Pascal string
PEA #'Entered main loop'; push address of string on stack
_DebugStr                ; enters MacsBug and displays message
```

When MacsBug gets control, it disassembles the instruction indicated by the PC and displays the contents of the registers. If the exception was caused by a $A9FF or $ABFF instruction, MacsBug displays the message "USERBRK", advances the PC to the next instruction, and then disassembles the instruction and displays the registers. It then displays the greater-than symbol (>) as a prompt, indicating that it is ready to accept a command.

Two other ways are available for entering MacsBug: FKEYs and INIT resources. By using ResEdit a skilled user can create a custom resource of either type whose sole function is described by two simple 68000 instructions: $A9FF $4E75 (_Debugger and RTS - the sequence to enter MacsBug)

Note: Another way to generate an exception that was popular in the past was to add a line such as

```
DC.W  $FECE       ; generate a line 1111 exception
```

at the point in your program where you wanted MacsBug to first get control. (Any value $F000 through $FFFF could have been used.) *This method should not be used any more*, as these instructions have been reserved by Motorola for use in their coprocessor interface for their 68020 microprocessor. (For example, in the future these "exceptions" could actually be MC 68881 floating point instructions!)

The MacsBug Command Language

Commands consist of one or two command characters followed by a list of zero or more parameters (depending on the command). Parameters can be numbers, text literals, symbols, or simple expressions.

Numbers

As is fitting for a debugger, *all numbers are hex unless otherwise specified*. Decimal numbers are preceeded by a pound sign (#). Hexadecimal numbers can optionally be preceded by a dollar sign ($). Numbers can be signed (+ or –). A hex word (4 hex characters) preceeded by a left angle (<) is sign extended to a long word. Here are some numbers in different formats—the formats shown are the same as those displayed by the CV (Convert) command, described later in this chapter.

Number	Unsigned Hex	Signed Hex	Decimal	
$FF	$000000FF	$000000FF	#255	
37	$00000037	$00000037	#55	
-FF	$FFFFFF01	-$000000FF	-#255	
#100	$00000064	$00000064	#100	
+10	$00000010	$00000010	#16	
#-32	$FFFFFFE0	-$00000020	#-100	
<FFFA		$FFFFFFFA	-$00000006	#-6

Strings

A text literal is a one- to four-character ASCII string bracketed by single quotes ('). If a string is longer than four characters, only the first four characters are used. When used by MacsBug, text literals are right justified in a long word. Here are some examples:

String	Stored as
'A'	$00000041
'Fred'	$46726564
'1234'	$31323334

Symbols

These symbols are generally used to represent the registers:

A0 through A7	Address registers A0 through A7
D0 through D7	Data registers D0 through D7
PC	Program counter
.	Last address referenced (“Dot”)
TP	Current QuickDraw port (thePort)

Expressions

Expressions are formed by operators acting on numbers, text literals, and symbols. The operators are

+	addition (infix); assertion (prefix)
-	subtraction (infix); negation (prefix)
@ or *	indirection operator (2 different prefix operators with identical functionality)
&	address operator (prefix)
<	add sign extended # (infix); sign extension (prefix)

The indirection operator uses the long integer at the location pointed to by the operand. Here are some valid expressions:

```
RA7+4
3A700-@10C
TP+#24
-RA0+RA1-'FRED'+@@4C50
RA5<FE34              (Same as RA5+FFFFFE34 - useful in looking at global variables)
```

Important: Expressions are evaluated from left to right. All operators are of equal precedence. There is no way to alter the order of evaluation.

Commands

MacsBug commands can be divided into five groups: memory, breaks, A-Traps, heap zone, and disassembly. A Return character repeats the last command entered, unless specified otherwise in the descriptions below.

Parameters are represented by descriptive words and abbreviations such as *address*, *number*, and *expr*. All parameters can be entered as expressions.

General Commands

? **(Help)**

Displays a short list of MacsBug commands and their parameters.

DV **(Display Version)**

Displays the version, the date and time of creation, and signature of MacsBug. For example,

```
MACSBUG 5.1B1  17-May-86 00:05:10  <DKA>
```

RB **(Reboot)**

Reboots the system.

ES **(Exit to Shell)**

Invokes the trap ExitToShell, which causes the current shell to be launched. The current shell is usually the Finder but can be changed by editing the boot blocks with FEdit. The current shell must reside in the System Folder and is logically different from the startup application.

Technical Note: ES may not work with applications that override important system traps. This is because the application heap gets initialized promptly upon calling the trap ExitToShell which usually trashes any system patches that used to be located there. However, there is a hook provided which is called by InitApplZone called IAZNotify that can be used to restore the world before purging the otherwise necessary routines.

EA **(Exit to Application)**

Relaunches the application. This is a faster method than calling ES and then relaunching from the Finder.

Memory Commands

CV *expr* **(Convert)**

Displays *expr* as unsigned hexadecimal, signed hexadecimal, signed decimal, text, and binary.

DM [*address* [*number*]] **(Display Memory)**

Displays *number* bytes of memory starting at *address*.

Number is rounded up to the nearest 16 bytes. If *number* is omitted, 16 bytes are displayed. If *address* and *number* are omitted, the next 16 bytes are displayed. The dot symbol is set to the address of the beginning of the last block displayed.

If *number* is set to certain four-character strings, memory is instead symbolically displayed as a data structure that begins at *address*. Refer to *Inside Macintosh* for a description of these data structures. The strings and the data structures they represent are

'IOPB'	Input/Output Parameter Block for File I/O
'WIND'	Window Record
'TERC'	TextEdit Record

You can usually terminate a DM command by pressing the Backspace key.

SM *address expr...* **(Set Memory)**

Places the specified values, *expr1* through *exprN*, into memory starting at *address*. The size of each value depends on the "width" of each expression. The width of a decimal or hexadecimal value is the smallest number of bytes that holds the specified value (four-byte maximum). Text literals are from one to four bytes long; extra characters are ignored. Indirect values are always four bytes long. The width of an expression is equal to the width of the widest of its operands. The dot symbol is set to *address* .

D*n* [*expr*] **(Data Register)**

Displays or sets data register *n*. If *expr* is omitted, the register is displayed. Otherwise, the register is set to *expr*.

A*n* [*expr*] **(Address Register)**

Displays or sets address register *n*. If *expr* is omitted, the register is displayed. Otherwise, the register is set to *expr*.

PC [*expr*] **(Program Counter)**

Displays or sets the program counter. If *expr* is omitted, the program counter is displayed. Otherwise, the PC is set to *expr*.

SR [*expr*] **(Status Register)**

Displays or sets the status register. If *expr* is omitted, the status register is displayed. Otherwise the status register is set.

TD **(Total Display)**

Displays all of the 68000 registers and the PC, and disassembles the current instruction that is about to be executed upon stepping, tracing, or going.

RX **(Register Exchange)**

Toggles the display mode so that the registers are or are not dumped during a trace command. The disassembly of the PC instruction is not affected.

CS [*address1* [*address2*]] **(Checksum)**

Checksums the bytes in the range *address1* through *address2* and saves that value. The checksum is an exclusive OR of the bytes in the range specified. If *address2* is omitted, it checksums 16 bytes, starting at

address1. If *address1* and *address2* are both omitted, it calculates the checksum for the last range specified, saves that value, and compares it to the previous checksum for that range. If the checksum hasn't changed, it prints "CHKSUM T"; otherwise it prints "CHKSUM F".

Break Commands

BR [*address* [*count*]] **(Break)**

Sets a breakpoint at *address*. *Count* specifies the number of times that the breakpoint should be executed before stopping the program. If *count* is omitted, the program is stopped the first time the breakpoint is hit. If *address* is omitted, all breakpoints are displayed. You can set a maximum of 8 different breakpoints.

CL [*address*] **(Clear)**

Clears the breakpoint at *address*. If *address* is omitted, all breakpoints are cleared.

G [*address*] **(Go)**

Executes instructions starting at *address*. If *address* is omitted, execution begins at the address indicated by the program counter. Control does not return to MacsBug until an exception occurs.

GT *address* **(Go Till)**

Sets a one-time breakpoint at *address*, then executes instructions starting at the address indicated by the program counter. This breakpoint is automatically cleared after it is hit. (GT *address* is equivalent to a BR *address* and G with the BR being cleared after the first time it is hit.)

T **(Trace)**

Traces through one instruction. Traps are treated as single instructions. If the next instruction to be executed is a JSR to a currently unloaded segment, you will see instead of the JSR the LoadSeg ($A9F0) trap. Tracing through that instruction will not work normally. If you wish to trace through the LoadSeg trap a low memory global needs to be set to a non-zero value. That location is $12D. Do a `SM 12D 1` to enable tracing through the LoadSeg call. Next, Go (G). You will break at an RTS instruction. Trace once (T) to see the absolute location that you are about to jump to. Trace again and you will be at the first step of the routine that now is loaded into memory.

S [*number*] **(Step)**

Steps through *number* instructions. If *number* is omitted, just one instruction is executed. Traps are not considered to be single instructions.

SS [*address1* [*address2*]] **(Step Spy)**

Calculates a checksum for the specified memory range, then does a Go; it then checks the checksum before each 68xxx instruction is executed, and breaks into MacsBug if the checksum doesn't match. If *address1* and *address2* are omitted, this feature is turned off.

Step Spy is very slow. Step Spy is nevertheless useful for detecting what routines are stepping on a specific place in memory. If checking memory every A-trap is sufficient for your needs, use the AS command, described below. (The slow motion capability of SS, however, can be useful in its own right to examine how the Finder zooms windows, for example. Think of it as a tool to study graphics algorithms.)

ST *address* **(Step Till)**

Steps through instructions until *address* is encountered. Unlike Go Till, this command does not set a breakpoint. Thus it can be used to step through, and stop in, ROM.

MR [*offset*] **(Magic Return)**

MR functions according to this formula:

IF *offset* > A6 THEN magic = *offset* + 4 ELSE magic = A7 + *offset*

The default *offset* is 0. This allows you to type MR RA6 when in nested subroutine calls. The usual way to use this routine is to trace until just after a JSR (return address 0 bytes down in the stack), and then do an MR. The rest of the routine is executed, and control returns to MacsBug. This command isn't repeated when you press Return; a Trace command is executed instead.

When debugging, you generally trace through a program one instruction at a time. MR lets you trace through to the end of a routine instead. When you use MR, it replaces the return address that is *offset* bytes down in the stack with an address within MacsBug; then it does a Go (described above). The RTS that would have used that address returns to MacsBug instead of to the caller. MacsBug restores the original return address, and then executes the RTS as if called by the Trace command. The prompt is then displayed, ready to trace the instruction after RTS.

DX **(Debugger Exchange)**

Normally, if either the $A9FF or the $ABFF A-trap (two forms of the Debugger trap) is executed, program execution halts and the debugger is activated. DX allows you to control whether or not program execution halts. Note that the $ABFF trap will still print a string; thus with debugger entry disabled an effect similar to that of the AT command occurs—that is, the Macintosh screen alternates between the debugger and the program. The default is to stop at Debugger traps.

A-Trap Commands

The A-Trap commands are used to monitor "1010 emulator" traps, used to call the Macintosh ROM. These commands use up to six parameters (*trap1, trap2, address1, address2, D1,* and *D2*). These parameters indicate which traps and other conditions should be monitored:

•*Trap1* and *trap2* specify the range of the traps. Operating System traps are in the range $A000 and $A0FF; Toolbox traps are between $A800 and $A9FF. Numbers in the range 0 to $6F are shortcut expressions for OS traps. Numbers greater than or equal to $70 are interpreted as Toolbox traps. If only *trap1* is specified, the command is invoked for *trap1*. If *trap1* and *trap2* are specified, the command is invoked for all traps in the range *trap1* through *trap2*. The defaults are $A000 and $AA00.

•*Address1* and *address2* specify a range of memory addresses within which traps should be monitored. The defaults are 0 and $FFFFFFFF.

•*D1* and *D2* specify the values of data register 0 within which traps should be monitored. They are treated as unsigned numbers. The defaults are 0 and $FFFFFFFF.

Thus if no parameters are given, all traps are monitored.

Unlike break commands, only one A-trap break command and one A-trap trace command can be active at the same time. In addition, the A-trap command with the "wider" range of traps must be executed last. For example, if you wished to view (trace) all file system traps called from your application but also wanted to stop at the next Open command, you could type the following commands:

```
BA   Open
AA   0 17            (shorthand for file system traps)
```

If you typed them in the reverse order you would not see any of the tracing because the smaller ranged command was executed last. In general, the parameters most recently given are the ones used; it is only the A-trap range that can be remembered for two different A-trap commands.

You generally use the Go command immediately after a trap command to await the use of a specified trap. When a trap in the indicated range is encountered, appropriate information is displayed. Displayed trap numbers are given in full word format ($Axxx).

BA [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]] **(Break in Application)**

Causes a break when the conditions specified by the parameters are satisfied and the trap is being called from the application rather than from the ROM. *Address1* and *address2* are automically set to ApplZone and BufPtr. Therefore this command can be used to get back to the application when in ROM. Simply type BA and Go. MacsBug will be entered at the next trap called by code located in the application heap. To break on ROM calls as well (or traps called from the System Heap or elsewhere) use AB which is described below.

AA [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]] **(Application A-Trap Trace)**

Traces and displays each A-Trap called from the application heap without breaking if the conditions specified by the parameters are satisfied. This command continues to display A-Traps until you press the interrupt button. This allows you to monitor just the traps that the application calls, and thus can be useful for checking and measuring performance. To monitor all traps called, including calls made from inside the ROM and traps called from the System Heap, use the AT command.

AB [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]] **(A-Trap Break)**

Causes a break when the conditions specified by the parameters are satisfied. AB without any parameters will thus stop at the very next trap executed anywhere by Macintosh. To stop at the next trap called by the current application, use BA instead.

AT [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]] **(A-Trap Trace)**

Traces and displays each A-Trap without breaking, when the condition specified by the parameters is satisfied. This command continues to display all A-Traps until you press the interrupt button. If you wish to just see the traps called by the current application, use AA instead.

For example, to see all QuickDraw calls displayed, regardless of who calls them, one could type

```
AT A86C A8FB
```

AH [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]] **(A-Trap Heap Zone Check)**

Checks the heap zone for consistency just before executing each trap in the specified range. If an inconsistency is found, it displays the addresses of the two memory blocks in question.

AR [*trap1* [*trap2* [*address1* [*address2* [*D1* [*D2*]]]]]] **(A-Trap Record)**

Whenever the parameter constraints are satisfied by an A-trap call, information about the call is recorded. The trap name, PC, A0, D0, and time are always saved. If the call was for an OS trap, 32 bytes pointed at by A0 are recorded; otherwise 32 bytes pointed at by A7 (stack ptr) are saved. To display the current saved information, type AR with no arguments.

This command is especially useful for tracking down crashes in the Macintosh ROM. For example,

```
>AR 0 1000 @2AA @114
```

records traps 0 through 1000 (all traps) from ApplZone ($2AA) through HeapEnd ($114), so it will record the last trap call made from anywhere in the application heap (the application's code).

AS *address1* [*address2*] **(A-Trap Spy)**

Calculates a checksum for the specified memory range, and then checks it before each and every A-trap that is called. Breaks into MacsBug if the checksum does not match. Use SS if you want the range of memory to be checked before every 68xxx instruction rather than just before every A-trap.

AX **(A-Trap Clear)**

Clears all A-Trap commands.

Heap Zone Commands

The heap zone commands act upon the current heap zone. When MacsBug is started up, the current heap zone is the application heap zone. You can set the current heap zone by using the HX command. Several commands cause MacsBug to scramble the heap zone. When MacsBug scrambles the heap zone, it rearranges all the relocatable blocks. This is useful for finding illegally used pointers to relocatable data structures.

HX [*address*] **(Heap Exchange)**

Sets the current heap to *address*. If no *address* is given, then HX toggles the current heap zone between the system heap zone and the application heap zone. In any case, HX displays the resulting current heap address.

HC **(Heap Check)**

Checks the consistency of the current heap zone, and displays the addresses of inconsistent memory blocks as well as the address of the current heap.

HS [*trap1 trap2*] **(Heap Scramble)**

Scrambles the heap zone, by moving relocatable blocks, when certain traps in the specified range are encountered. It always scrambles the heap zone as a result of NewPtr, NewHandle, and ReallocHandle calls. It scrambles the heap zone as a result of SetHandleSize and SetPtrSize if the new length is greater than the current length. This command is fastest if you set *trap1* to $18 and *trap2* to $2D. The heap zone is not scrambled as a result of traps other than those named above.

HD [*mask*] **(Heap Dump)**

Mask is optional. Whether or not *mask* is used, it displays each block in the current heap zone in the following form:

blockAddr type size [*flag MP_location*] [*] [*refNum ID type*]

•The *blockAddr* points to the start of the memory block.

•The *type* is one of the following letters:

F	free block
P	pointer
H	handle to a relocatable block

•The *size* is the physical size of the block, including the contents, the header, and any unused bytes at the end of the block.

•For handles (type H), *flag* is either blank if not purgable or a 'P' if purgable. Then *MP_location* is displayed, which is the address of the master pointer to the relocatable block.

•The asterisk marks any locked object (nonrelocatable blocks and locked relocatable blocks).

•For resource file blocks, three additional fields are displayed: the resource's reference number, ID number, and type. If *mask* is omitted, the dump is followed by a summary of the heap zone's blocks. It begins with the six characters "HLP PF", which represent the six values that follow them. These values are

H	number of relocatable blocks in the heap zone (Handles)
L	number of relocatable blocks that are Locked
P	number of Purgeable blocks in the heap zone
(space)	space, in bytes, occupied by purgeable blocks
P	number of nonrelocatable blocks in the heap zone (Pointers)
F	total amount of Free space in the heap zone

Here is a sample summary:

```
HLP PF 0084  0004  0002  0000079E  0017  000003B4
```

Note that block counts are single words, and values representing space in bytes are long word quantities. If *mask* is used, the summary line displays the block counts of specific types of blocks. Possible values for *mask* are

'H'	Relocatable blocks (handles)
'P'	Nonrelocatable blocks (pointers)
'F'	Free blocks
'R'	Resource blocks

'xxxx' Resource blocks of type 'xxxx'

If *mask* is used, the heap summary takes this form:

CNT ### <# *of blocks of mask type*> <# *bytes in those blocks*>

You can prematurely terminate an HD command by pressing the Backspace key.
The dot address is set to the last block of memory displayed by HD.

HT [*mask*] **(Heap Total)**

Displays just the summary line from a heap zone dump. *Mask* works just as it does with the HD command (described above).

SC **(Stack Crawl)**

Assumes that LINK / UNLK A6 has been religiously performed at the beginning and end of each procedure or function. (Use the $D+ option in Pascal, the -g and -ga options in C) The output format is as follows:

SF *@<stack frame location>* *<address of call to procedure>*

For example,

```
SF @0D633C  ProcName+3A
```

means that the currently executing procedure or function has its local stack frame at $D633C and was called from ProcName+$3A (which is not the return address!). As the ROM does not use LINK / UNLK consistently (due to space & time considerations), SC does not work when in ROM routines.

Disassembler Commands

SD [*address*] **(Symbol Dump)**

Displays a list of the procedure names that can be found in the current heap zone. The search criteria is based on looking in blocks of memory whose locked bit is set. In addition, a LINK A6 and its matching UNLK A6 must be found, followed by either a JMP (A0) or a RTS. The eight character debugging name follows. Valid debug symbols must have characters between an ASCII space ($20) and the underscore character ($5F) inclusive. This command optionally allows you to specify a starting location for the symbol dump.

PX **(Symbol Toggle)**

Determines whether or not symbols are displayed. By default, symbols are turned on. This command affects any command that takes an address. Using symbols allows you to IL or BR on a procedure name. For example,

```
>IL ProcName+58
```

will disassemble code starting at 58 bytes (hex) into the procedure called ProcName, and

```
>BR ProcName+58
```

will set a breakpoint at the same location (this also works for GT, ST, DM, and so on.).

Note that your code must be created with the symbols embedded in the code in order for this to work.

DH *number* **(Disassemble Hex)**

Disassembles the hex byte, word, or long word input. Typing just one byte allows you to see the general class of instructions, as *number* is left aligned in a long word padded to the right with zeros. (Typing `DH 10`, `DH 20`, and `DH 30`, for example, show by induction that these instruction groups are the Move.B, Move.W, and Move.L classes, respectively.)

This command is useful as a poor man's assembler: guess an opcode and use this command to see if it is what you want. Example: was reset $4E71? Type `DH 4E71` and see NOP. Try `DH 4E70` and see RESET.

ID [*address*] **(Instruction Disassemble)**

Disassembles one line at *address*. If *address* is omitted, the next logical location is disassembled. ID sets the dot symbol to the *address*.

If the code has symbols compiled in with it, and symbols have been turned on with the PX command, each address is automatically displayed as a routine name plus an offset.

IL [*address* [*number*]] **(Instruction List)**

Disassembles *number* lines starting at *address*. If *number* is omitted, a screenful of lines (typically 16) is disassembled. If both *number* and *address* are omitted, a screenful of lines is disassembled starting at the next logical location. This command sets the dot symbol to the *address*.

If the code has symbols compiled in with it, and symbols have been turned on with the PX command, each address is automatically displayed as a routine name plus an offset.

You can prematurely terminate an IL command by pressing the Backspace key.

F *address count data* [*mask*] **(Find)**

Searches *count* bytes from *address*, looking for *data* after masking the target with *mask*. As soon as a match is found, the *address* and value are displayed, and the dot symbol is set to that *address*. To search the next *count* bytes, simply press Return. The size of the target is determined by the width of *data*. It is limited to 1, 2, or 4 bytes.

For example, to find a RESET instruction in a program loaded into a 1 MB machine, you could type

```
>F CB00 EFFFF 4E70
```

where CB00 is the beginning of the application heap on a Macintosh Plus, EFFFF represents the length of the application heap (roughly), and 4E70 is the RESET instruction.

WH *expr* **(Where)**

Where takes an expression, which can be a symbolic name. It displays the location of the first routine that it finds whose name matches. ROM symbol names are ten character names, while RAM symbols are eight character names.

If *expr* is less than $AA00, this command displays the address corresponding to the trap with that *number*. All of the following commented commands, for example, give the same result:

```
>WH     EXITTOSHELL   ; full name
>WH     A9F4          ; full trap word
>WH     1F4           ; shortcut
>WH     40F6D8        ; address of ExitToShell in the 128KB ROM
```

Namely,

Trap Word	**Address**	**Name**
A9F4	40F6D8	EXITTOSHELL

The shortcut method of inputting trap numbers interprets $0-$6F as OS traps and all other traps as Toolbox traps.

If *expr* is preceeded by the address operator (&) then the expression is forced to be evaluated as an address. This is useful for examining system patches whose addresses are oftentimes less than $AA00, the default address boundary.

If *expr* is greater than or equal to $AA00 and less than RomBase, then the address is interpreted as a user routine in RAM, and a symbolic location will be displayed if possible.

If *expr* is in ROM then the trap whose code is closest to that address is displayed.

This function is useful for finding out where you were when an error occurred. If the address expression is in RAM and the WH function returns “PRGM AT $$$$” you can then use the HD 'CODE' command to list the code segments. Then by comparing the locations of CODE segments and the current PC you can determine which segment you are in.